Array-Structured Object Types for Mathematical Programming

Felix Friedrich and Jürg Gutknecht

Computer Systems Institute, ETH Zürich, Switzerland
{felix.friedrich, gutknecht}@inf.ethz.ch

Abstract. In this paper a concept for structured mathematical programming within an object-oriented language is presented. It leads to better readable, more natural and more compact code in typical linear algebra applications and provides options for optimized implementation. We also discuss the realization of this concept as an extension of the programming language Active Oberon.

We define new built-in array types that provide a slight modification of classical arrays in Oberon. By introducing range-valued indices as array designators, we permit the use of regular sub-domains of arrays as parameters of operators and procedures. The built-in types are complemented by *custom* array structured object types. The latter can be specified by the programmer and are designed to be syntactically compatible with the former. They provide the needed flexibility for the language.

1 Introduction

There are already concepts for mathematical programming proposed both in multi-purpose languages, such as Fortran, Zpl and Chapel, and in special purpose packages like Matlab, Mathematica and R, just to mention a few. Concepts in common mathematical languages are too specific and functionality is too complex for a general purpose language. However, these approaches must not be ignored but rather be used for inspiration.

Besides other advantages of the programming language Oberon, its clarity and readability is undoubtedly a good reason to go for it. Because it is more abstract and closer to mathematics than system near languages such as C/C++, Java etc., it permits to implement mathematical algorithms in very clear and structured form. However, by experience and inspection of code, in particular for linear algebra and imaging applications, we discovered that a small extension of the language can significantly increase efficiency and readability.

We cannot present a solution that satisfies all possible needs. Although it is tempting to implement as much functionality as possible, we aim at a coherent, self-contained concept that avoids redundant language constructs and programming pitfalls. To achieve such a lean model, we attach equally much importance to constructs that we provide and to functionality that we deliberately *omit*.

We believe that a programmer can enhance an implementation considerably without having to deal with system near constructs: The ideal case is of course the development of theoretically better algorithms providing lower complexity and lower run-times. But also using the structure inherent to a problem in the implementation can be of high value. For instance, in the context of array operations, existing code can be made considerably clearer and more efficient by exploiting that certain operations can be performed block-wise. We are not aiming at an automatic enhancement on the code generation level (as, for instance, in ATLAS, cf. [2]) but want to give the programmer tools at hand with which he can incorporate his expert knowledge about the structure of the matter.

The objective of this paper is to establish an object oriented concept of (multi-dimensional) array-structured types. Purpose is intuitive and efficient mathematical programming. The paper is organized as follows: Section 2 has a motivating nature. It provides some preliminary examples of our language extensions and contains conceptional considerations. The new language constructs are then presented in Section 3 in detail. This last part comprises the formal specification of built-in and custom array types, of operators on and between them, of range-valued indices used as array designators and some implementation specific notes. The paper ends with a conclusion.

2 Preliminary Conceptual Considerations

The first part of this section contains examples providing a quick insight to our new language constructs. In the second part we will give reasons for the design that is then particularized in Section 3. The current state of the art in Oberon is recapitulated in the third part.

2.1 Illustration

In this paragraph, we examine code from a typical Oberon linear algebra implementation and illustrate our approach by ways of these examples. The examples are not exhaustive and anticipate notions that will be explained in Section 3.

Operators: Matrix Multiplication. The most prominent example of a linear algebra operation is certainly the multiplication of two matrices. A naïve Oberon version is displayed in Fig. 1 and a version with elimination of the inner loop is depicted in Fig. 2. It is obvious that, having readability in mind, this construct in general has to be replaced by a call to a procedure or better by a language-integrated multiplication operator as displayed in Fig. 3. But not only this can be learned from the displayed algorithm: If `L` and `R` are large matrices then cache misses are highly probable in the inner loop, since `R` is processed column-wise. Operators between arrays and dimensions-permuted storage formats are of benefit here and possible in the new approach.

```
VAR A,B,Res: POINTER TO ARRAY OF ARRAY OF REAL;
    i,j,k: LONGINT;
    temp: REAL;
(* ... *)
(* check shapes *)
FOR i := 0 TO LEN(L,0)-1 DO
 FOR j := 0 TO LEN(R,1)-1 DO
  temp := 0;
  FOR k := 0 TO LEN(R,0)-1 DO
   temp := temp + L[i,k]*R[k,j];
  END;
  Res[i,j] := temp;
 END;
END;
```

Fig. 1. Naïve matrix multiplication

```
VAR A,B,Res: POINTER TO ARRAY OF ARRAY OF REAL;
    i,j: LONGINT;
(* ... *)
(* check shapes *)
FOR i := 0 TO LEN(L,0)-1 DO
 FOR j := 0 TO LEN(R,1)-1 DO
  Res[i,j] := L[i,..]+*R[..,j]; (* pseudo scalar product *)
 END;
END;
```

Fig. 2. Naïve matrix multiplication,inner loop eliminated

```
VAR A,B,Res: ARRAY [..,..] OF REAL;
(* ... *)
Res := A*B;
```

Fig. 3. Matrix multiplication with natural notation

Sub-array Structures. Very often operations are not performed on the complete array but rather on sub-array structures, such as (parts of) columns or rows of a matrix. A first example with operation on rows and columns of a matrix has already been displayed in Fig. 2.

The singular value decomposition algorithm provided by LAPACK is one prominent example consisting of many such operations. In Fig. 4 a small portion of the code is displayed. Our approach includes range-valued indices that, applied to an array, form a designator of certain substructures, see Fig. 5.

```
VAR u: POINTER TO ARRAY OF ARRAY OF REAL;
    s,h,f: REAL; i,j,k,l,m,n: LONGINT;
(* ... *)
FOR j := l TO n DO
 s := 0.0;
 FOR k := i TO m DO
  s := s + u[k, i] * u[k, j]
 END;
 f := s / h;
 FOR k := i TO m DO
  u[k, j] := u[k, j] + f * u[k, i]
 END;
END;
```

Fig. 4. Small part of SVD in classical notation

```
VAR u: ARRAY [..,..] OF REAL; s,h: REAL; i,j,l,m,n: LONGINT;
(* ... *)
FOR j := l TO n DO
 S := u[i..m,i]+*u[i..m,j];            (* scalar product *)
 u[i..m,j] := u[i..m,j] + s/h* u[i..m,i]; (* element-wise operations *)
END;
```

Fig. 5. Code from Fig. 4 using new approach

Custom Array Types. Since not all possible features can be implemented in a built-in array type, we have made provision for the implementation of custom array types. Figure 7 contains a sample implementation of a sparse matrix, i.e. a two dimensional array that only has a small number of nonzero elements. In Fig. 6 it is shown how such a new type harmonizes with the concept of 'normal' arrays. Note that the two dimensional array structure and the element type is constituted in the (array) type declaration of `SparseMatrix`.

```
VAR A: ARRAY [10,10] OF REAL; B: SparseMatrix; i: LONGINT;
 (* ... *)
 A := 1;                      (* fill matrix A with ones *)
 NEW(B,1000,1000);            (* sparse matrix of size 1000x1000 *)
 FOR i := 0 TO 999 BY 10 DO
  B[i..i+9,i..i+9] := A;      (* fill blocks along diagonal *)
 END;
```

Fig. 6. Using custom array types. Implementation of `SparseMatrix` suggested in Fig. 7

```
TYPE
 SparseMatrix*= ARRAY [..,..] OF REAL (* 2d array structure with element type real *)
  VAR d: Data; len0,len1: LONGINT; (* assume type Data is defined elsewhere *)

  PROCEDURE NEW(i,j: LONGINT); (* allocation *)
  BEGIN
   (* create data structure *)
   len0 := i; len1 := j;
  END NEW;

  PROCEDURE LEN(i: LONGINT): LONGINT; (* sizes, shape *)
  (* ... *)
  END LEN;

  PROCEDURE "[]"(i,j: LONGINT): REAL;
  BEGIN
   (* range check *)
   RETURN Get(d,i,j)
  END "[]";

  PROCEDURE "[]"(i,j: LONGINT; r: REAL);
  BEGIN
   (* range check *)
   Put(d,i,j,r);
  END "[]";

  (* matrix extraction *)
  PROCEDURE "[]"(a1..b1 BY c1,a2..b2 BY c2: LONGINT): ARRAY [..,..] OF REAL;
  VAR A: ARRAY [..,..] OF REAL; (* in this implementation: extract block as built-in array *)
  BEGIN
   IF a1 = MIN(LONGINT) THEN a1 := 0 END;  (* defaults *)
   IF b1 = MAX(LONGINT) THEN b1 := len0-1 END; (* defaults *)
   (* same for a2,b2 *)
   (* range check *)
   NEW(A,(b1-a1) DIV c1,(b2-va2) DIV c2);
   Extract(d,A,a1..b1 BY c1; a2..b2 BY c2);
   RETURN A;
  END "[]";

  (* submatrix assignment *)
  PROCEDURE "[]"(a1..b1 BY c1,a2..b2 BY c2: LONGINT; VAR A:ARRAY [..,..] OF REAL);
  BEGIN
   (* defaults, range check *)
   Insert(A,d,a1..b1 BY c1, a2..b2 BY c2);
  END "[]";

 END SparseMatrix;

 (* Get, Set, Extract, Insert routines skipped *)

 (* operator overloading *)
 PROCEDURE '*' (A,B: SparseMatrix): SparseMatrix;
 (* ... *)
 END '*';

 PROCEDURE '*' (A: SparseMatrix; VAR B: ARRAY [..,..] OF REAL): ARRAY [..,..] OF REAL;
 (* ... *)
 END '*';

 (* ... *)
```

Fig. 7. Draft of a sparse matrix implementation

2.2 Design Objectives

In this paragraph we state basic conditions and establish a concept that is in compliance with them.

Requisites. Our goal is an approach that, in particular for arrays, supports the following general key requirements.

1. *Efficiency.* It should be possible that expert knowledge about the structure of an algorithm is incorporated into an implementation to achieve efficiency.
2. *Notational simplicity.* Mathematical programs must well be readable and notation should conform with usual mathematical conventions.
3. *Structural simplicity.* A programmer must not need to handle system matters like complicated pointer arithmetics and memory management.
4. *Extensibility.* The built-in features of a language cannot satisfy all possible needs. It should thus be possible to add arbitrary functionality on an implementation level if it agrees with stipulated syntax and semantics.
5. *Safety.* Typical safety features, such as range- and type-checking must be preserved by the extension of the language.

To achieve efficient implementations of array-based algorithms, fast single element accesses are obviously necessary in the first place. Also the availability of optimized block-wise operations on sub-array configurations can improve speed considerably in many cases. The most prominent example is the generalized matrix multiplication identified to be the main performance kernel of the Basic Linear Algebra Subprograms (BLAS), cf. [2], p. 10. Figure 8 illustrates the gain of speed reached by using an optimized matrix multiplication using Intel's Streaming SIMD extensions (SSE), which add vector-oriented capabilities to general purpose processors. The displayed measurements refer to inline assembler code within an optimized Oberon module. Optimizations of this and similar kind will be done by the compiler and can in principle be applied to any type of regular array substructure. A discussion of optimization techniques in detail is beyond the scope of this paper.

Another important issue for efficiency is the avoidance of cache missing and cache trashing when dealing with large data, cf. [7]. Notational simplicity implies that block-wise operations have to be denoted in a common form and that specific optimizations, such as the avoidance of cache missing, must happen behind the scene and should not affect the notation. In the context of array handling, extensibility implies the implementation facility of arrays that cannot be represented as a linear piece of memory. Typical examples are sparse matrices or images with special boundary conditions such as 'periodic', 'mirrored' etc. Regarding safety, array range checks are indispensable as they are substantial for debugging and vital for system safety.

2.3 Concepts of the New Array Types

To comply with the aforementioned requisites, we decided to extend the functionality of Oberon *built-in* arrays and complement them by compatible

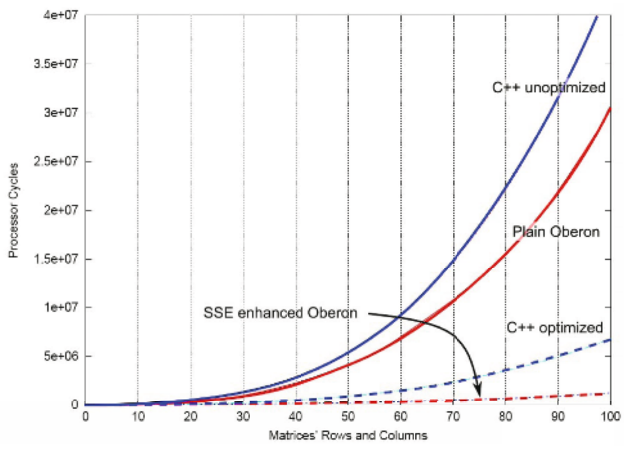

Fig. 8. Processor cycles of matrix multiplication. Oberon vs. C++ vs. optimized C++ vs. Oberon using SSE instructions, source: [14].

(programmer-definable) *custom* array-structured types. To explicitly discriminate the newly proposed built-in arrays from the classical array types in Oberon, we will in this text now and then denote them as *special* arrays. Special arrays permit the use of ranged indices as array designators. This construct allows to pass regular sub-domains of arrays to procedures and to use them as operands in expressions. This, together with the availability of efficient operators, leads to more readable and efficient code in linear algebra applications. Moreover, it allows the identification of independent pieces of code that can be optimized, for example being executed in parallel. As a further positive side effect, the needed array memory representation permits a dimension-permuted storage scheme that can be utilized for the avoidance of cache missing. Range checks are performed for each single element access and can be optimized to one single check for the access to an entire sub-structure. Safety is thus preserved while efficiency can be achieved by using the concept of ranges and operators.

For mathematical programming we generally prefer value semantics to reference semantics as it assures unambiguity of operations, in particular assignment and test for equality. Consequently, special arrays are value types (like records), rather than reference types (like objects). Memory allocation and pointer mechanisms are performed on behalf of the programmer behind the scenes. The programmer is only confronted with the definition and usage of fix- or variable-sized arrays. As a consequence, dynamic arrays are not exposed as pointers to an array structure: an array may well be of length zero but physically it invariably consists at least of the descriptor containing information about its shape. The decision for value semantics does not imply a severe restriction since arrays may still be wrapped into records or objects. For shared access this would be necessary anyway since concurrent access is managed by mutual exclusion on an object level in Oberon. Value semantics can also be regarded as additional protection against unintentional concurrent access to an array.

Custom array types can be specified by the programmer and are designed to be syntactically compatible with the built-in arrays. They provide the needed flexibility for the language. For operations on and between array types we use the already implemented operator concept of Oberon together with the facility of overloading. Internally, a custom array type is designed like a *value object type whose signature explicitly contains the array structure*. In this respect it is not regarded as an extension of a built-in array, but merely as a custom type that *mimics* the behavior of an array. Custom array types are abstract data types that may contain variables and procedures, but cannot be extended. The most important difference to the indexer concept of C# is that the array access structure of a custom array type is provided and fixed by its signature. We regard array structure as not only a property but merely as very immanent feature that must be statically tied to the respective object. In particular it allows to define substructures of object types in a clean way and prevents the misuse of mathematical 'indexers' for general purposes.

The dimension of an array is statically determined, i.e. cannot be changed at runtime, neither for built-in types nor in the programmer-definable form.

We repeat the main achievements of this concept. It permits

1. notational compactness in linear algebra applications,
2. optimizations by utilizing block-wise operations while preserving safety,
3. a clean implementation of (non-contiguous) custom array structures.

Discarded Ideas. Arrays in general stand for data of the form E^S, where E is a set of possible single states and S is a subset of $\mathbb{Z}^d$. Thus the specification of an array type `A` requires the definition of an element type `B` (referring to the set of single states E), a specification of the index set S and access patterns for elements of `A`. Although it somehow reflects the mathematical nature of E^S, for the sake of simplicity we do not introduce a separate type for the domain S as for example done in the programming language Chapel [4] and (partially) in ZPL [5].

The following features are of interest in some applications and can be implemented with custom array types. For built-in types, however, we decided against them: Customizable lower bounds for arrays provide potential pitfalls in programming, therefore built-in arrays have a fixed lower bound of zero. Free boundary conditions, such as "mirrored", "wrapped" etc., cannot be set for the built-in arrays, because it would generally prohibit efficiency for single element access. Built-in arrays do not permit the appending, insertion and deletion of elements since this requires a complex data type. (A reasonable implementation is provided by the software package Voyager, cf. [16]). The same holds for a built-in type of a sparse array representation. Having a common type for both 'normal' and sparse arrays would represent a dilemma for efficiency. Moreover, there are various forms of matrix storage schemes, such as Compressed Row/-Column Storage, Jagged Diagonal Storage etc., cf. [8]. We therefore decided to provide the flexible and efficient sparse matrix specification according to [3] as *sample implementation* using custom array-structured types.

Another approach that we discussed was the support of *properties* (built-in attributes) of arrays / matrices such as ‘diagonal’, ‘symmetric’ etc. on a language level. For example, the (dynamic) array structure could be taken into account to optimize the execution speed of operators like multiplication. On a static level this is already possible using custom array types. However we decided that the rare cases where a dynamic optimization would be possible are not worth the enormous computational effort and discarded this idea.

We also discarded the uses of indexers, as for instance provided by C#, because taking substructures would not be possible in a clean way, compare previous paragraph.

3 Specification of the Language Extension

In this section the syntax of the new built-in arrays and custom array types in Oberon is provided. Further some implementation specific notes are stated. We first recapitulate the status quo of Oberon: In classical Oberon it is not possible to address sub-arrays that do not form a contiguous block. New array types with different element access rules, such as sparse matrices, cannot be added to the system. The dimension order in memory coincides with that of the notation. From the view of mathematical programming the pointer notation used in Oberon for dynamic arrays is somewhat unnatural.

3.1 Built-In Arrays

Special arrays do not replace the classical arrays of Oberon but are added to the language. A special array type is determined by a statement that is compliant with the EBNF rule

```
ARRAY "[" Length{"," Length} "]" OF Type ";".        (1)
```

where `Length` is either given by an expression or two periods:

```
Length = ".." | Expression.        (2)
```

The index set of an array is a rectangular d-dimensional set. The lower bound is zero in each dimension. Special arrays can be defined statically, semi-dynamically, dynamically and open. They are regarded as value types. Unallocated dynamic arrays have zero length dimensions. Constant arrays can also be specified like displayed in Fig. 9.

Operators. There are unary and binary operators predefined. Binary operators apply to two arrays or an array and a base type. Most important is the operator ‘`:=`’: Special arrays may be assigned to each other. Since they are of value type, assignment infers copy of content. For any operation on two arrays with a compatibility requirement, such as assignment, the shape of the arrays must

```
A: ARRAY [..,..] OF REAL (* declaration of dynamic size matrix *)
B: ARRAY [3,5] OF REAL  (* declaration of static size matrix *)
LEN(B,i)                  (* length of dimension i, LEN(B)=LEN(B,0) *)
NEW(A,3,5)                (* allocation of dynamic size matrix *)
[[1,2,3],[4,5,6]]         (* constant array *)
r := A[i,j]               (* element read access *)
A[i,j] := r               (* element write access *)
```

Fig. 9. Some examples regarding the notation of special arrays

match, that is `dim(A)=dim(B)` and `LEN(A,i)=LEN(B,i)` for all $0 \leq i <$ `dim(A)`, and the element types must be compatible w.r.t. the operation. The predefined operators on and between arrays are displayed in Fig. 10 and 11. With respect to efficiency, the objective of having operators is to leave open the possibility of fast (potentially parallel) execution of operations that usually require many single element accesses. The displayed operators (together with the special cases for matrices, see below) are chosen from typical applications in linear algebra and are promising with respect to significant speed-up of most frequently used routines. The notation is deliberately designed to be near to that of MatLab.

operator	operand	result	meaning
'-'	array of number	array	element-wise negation
'~'	array of boolean	array	element-wise inversion
'ABS'	array of number	array	element-wise absolute value
'MIN' , 'MAX'	array of number	scalar	minimal and maximal element
'SUM'	array of number	scalar	sum of elements
'PRODUCT'	array of number	scalar	product of elements

Fig. 10. Unary array operators

operator	operands	result	meaning
':='	scalar,array	array	assignment of value to each element
':='	array,array	array	assignment of same sized arrays
'+' , '-' , '*'	array,scalar	array	element-wise scalar operation
'/' , 'MOD' , 'DIV'			
'+' , '-' , '.*'	array,array	array	element-wise operation
'/' , 'MOD' , 'DIV'			
'+*'	array,array	scalar	pseudo scalar product
'='	array,array	boolean	test of equality

Fig. 11. Binary array operators

For arrays of non-arithmetic types, the operators are still undefined (but can be overloaded). The definition of the (pseudo-) scalar product `A +* B` is necessary for performance reasons: `SUM(A .* B)` requires array allocation while

```
VAR A,B: ARRAY [..,..] OF REAL; r: REAL; b: BOOLEAN;
(* ... *)
B:= -A;       (* element wise negative of A *)
B:= ABS(A);   (* element wise absolute value of A *)
MIN(A), MAX(A) (* minimal / maximal element of A *)
A + B, A - B  (* sum and difference *)
A .* B, A./ B (* element-wise product and quotient *)
A +* B        (* (pseudo) scalar product *)
b := A=B;     (* equality *)
```

Fig. 12. Operators on and between special arrays

A +* B does not. Examples regarding notation of operators are displayed in Fig. 12.

Special Case: Matrix Operators. According to [1] and [2], the most important and time-critical operation in the Basic Linear Algebra Subroutines (BLAS) package is the one for generalized matrix multiplication. Moreover, the solution of matrix equations as displayed below is also a prominent example, again the notation follows MatLab. The following operators are defined for two dimensional arrays. The unary operator "'" does not create a copy of the data but only a designator to the same array with toggled dimensions. This is possible due to the internal format of the array references, cf. paragraph 3.3. Examples are given in Fig. 14.

Remark: "/" and "\" will not necessarily be provided as built-in operators.

operator	operands	result	meaning
' (postfix)	2d array	2d array	transposed of matrix
*	2d array,2d array	2d array	matrix product
/	2d array,2d array	2d array	solution of equation system
\	2d array,2d array	2d array	solution of equation system

Fig. 13. Operators on two dimensional arrays

```
VAR A,B,C,X: ARRAY [..,..] OF REAL;
C := A * B; (* matrix product *)
X := B / A; (* solution of equation X*A=B, read: B*A^(-1) *)
X := A \ B; (* solution of equation A*X=B, read: A^(-1)*B *)
A := B';   (* B' is reference to transposed of B, copy by ":=" *)
```

Fig. 14. Operators on and between two dimensional arrays

Ranges. A range is denoted by an expression of the form

`[Expression]..[Expression][BY Expression].` (3)

Consider the range `a..b BY c`. Here the symbols `a`, `b` and `c` (`c` > 0) must be integer valued constants or integer variables. The range `a..b BY c` stands for the set

$$\{a + i \cdot c : i \in \mathbb{N}, 0 \leq i \cdot c \leq b - a\}.$$

The usage of this notation is limited to the call and declaration of procedures and of the index operators `'[]'`. If `c` is not specified, then a value of 1 is assumed. If `a` or `b` is not given, then – depending on the context – the smallest or largest appropriate value is imputed. If not specified but explicitly referred to, a value of `MIN(LONGINT)` and `MAX(LONGINT)` is presumed on `a` or `b`, respectively. For instance the call `TestRange(..)` of the procedure

```
PROCEDURE TestRange(a..b BY c: LONGINT)
```

results in `a=MIN(LONGINT)`, `b=MAX(LONGINT)`, `c=1` in the procedure body.

Ranges on Arrays. Ranges can be applied to special arrays. A variable specified by the expression

```
Identifier[Range|ConstExpr{,Range|ConstExpr}]          (4)
```

is formally of array type with dimension equal to the number of ranges given. It is a designator and is therefore not necessarily materialized but only stands for a certain part of the array. As in the case of ordinary indices, a designator is applicable for read and write access.

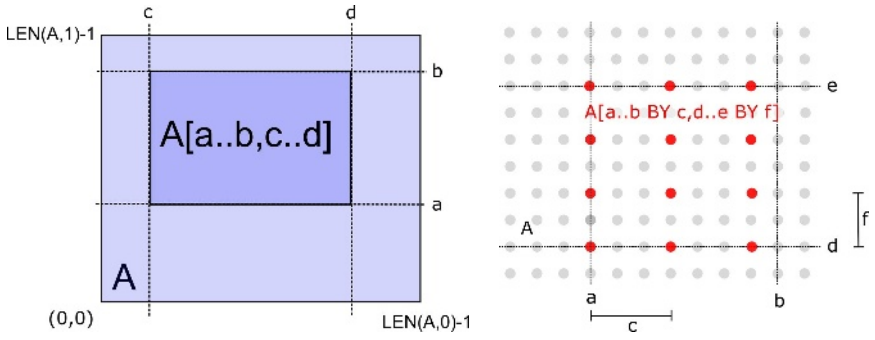

Fig. 15. Illustrations of domain extraction

Note that there is a substantial difference between a number `i` and a range `a..b BY c` in the specification of a sub-domain. `A[a..b,c..d]` stands for a two dimensional array, even if `a=b` or `c=d`, while `A[a..b,i]` stands for a one dimensional array and `A[i,j]` stands for a number.

Sub-domain specifications, such as `A[a..b,c..d]` refer to the same data as the referenced object (`A`). So referring to `A[a..b,c..d]` in the procedure declaration with a VAR parameter allows to modify the content of A. However, the statement sequence

```
B := A[..,..]; B[2,2]:= r
```

does not modify the content of A since the assignment operator ':=' stands for copy operation. More examples are displayed in Fig. 16.

```
VAR V: ARRAY [..] OF REAL; A: ARRAY [..,..] OF REAL;
V[..10] (* stands for V[0..10], is of type ARRAY [..] OF REAL *)
V[3..] (* stands for V[3..LEN(A)-1] *)
V[.. BY 2] (* stands for V[0..LEN(A)-1 BY 2] *)
A[a1..b1 BY c1, a2..b2 BY c2] (* two dimensional subdomain of A *)
A[a1..b1,a2..b2] := [[1,2,3],[4,5,6],[7,8,9]]; (* assignment of const *)
V := A[a1..b1,a2]; (* copy of piece of column *)
V := A[a1,a2..b2]; (* copy of piece of row *)
(* assume PROCEDURE MyProc(v: ARRAY [..] OR REAL); *)
MyProc(A[..,5]); (* call procedure, pass 6th column of A as parameter *)
```

Fig. 16. Examples regarding use of ranges

3.2 Custom Array Types

Besides the built-in functionalities, provision is made for the free specification of structured array types and operators. In this paragraph the syntax and semantics are defined.

Definition of Custom Array-Structured Types. A custom array type may be defined by the programmer like an object type. Inheritance and polymorphism is not supported. Moreover, custom array types cannot have an (active) body in Active Oberon. The reason for this decision is clearness of the language definition: built-in array types and custom array types must be handled equivalently and the atomic evaluation of operators is not guaranteed for the first. Synchronization has to be done on an object level if references are used for the arrays. As mentioned previously, this is additionally ensured by the value semantics used.

A custom array is specified with the pattern

```
TYPE ident "≟" ARRAY "[" ..{,..} "]" OF Type   DeclSeq   END ident.
```
(5)

The minimal ingredients that are usually implemented are the procedures NEW, LEN and read- and write-access methods "[]" as displayed in Fig. 17.

Ranges on Custom Array Types. For custom array-structured types, typically the procedures depicted in Fig. 18 would be implemented to obtain range accesses. The compiler discriminates between different forms of '[]' by their different signatures. Generally, only the array specific operators LEN and the index operators are directly declared within array scope whereas other operators have to be declared outside in module scope.

```
TYPE SparseMatrix = ARRAY [..,..] OF REAL
VAR (* ... *) (* allocation variables etc. *)
  PROCEDURE NEW(i,j: LONGINT); (* initialization, allocation *)
  PROCEDURE "[]"(i,j: LONGINT): REAL; (* read access *)
  PROCEDURE "[]"(i,j: LONGINT; r: REAL) (* write access *)
  PROCEDURE LEN(i: LONGINT): LONGINT; (* shape *)
END SparseMatrix;
```

Fig. 17. Custom array type definition I

```
TYPE
Matrix= ARRAY [..,..] OF REAL;
Vector= ARRAY [..] OF REAL;

SparseMatrix = ARRAY [..,..] OF REAL
  ...
  (* read access routines *)
  PROCEDURE "[]"(a1..b1 BY c1, a2..b2 BY c2: LONGINT): Matrix;
  PROCEDURE "[]"(a1..b1 BY c1, i: LONGINT): Vector;
  PROCEDURE "[]"(i, a2..b2 BY c2: LONGINT): Vector;
  (* write access routines *)
  PROCEDURE "[]"(a1..b1 BY c1, a2..b2 BY c2: LONGINT; VAR A: Matrix);
  PROCEDURE "[]"(a1..b1 BY c1, i: LONGINT; VAR A: Vector);
  PROCEDURE "[]"(i, a2..b2 BY c2: LONGINT; VAR A: Vector);
END SparseMatrix;
```

Fig. 18. Custom array type definition II

Operators. Generic operators can also be defined for custom array types. As mentioned, operators must be defined within the array type module scope. At least one of the operands must be part of the current scope. The definition of overloaded operators follows the current Active Oberon convention. For the `SparseMatrix` example, operators would typically be defined as in Fig. 19.

```
PROCEDURE ":=" (VAR dest: ARRAY [..,..] OF REAL; src: SparseMatrix);
PROCEDURE "*" (src1,src2: SparseMatrix): ARRAY [..,..] OF REAL;
PROCEDURE "*" (l: SparseMatrix; r: REAL): SparseMatrix;
PROCEDURE "+*" (l,r: SparseMatrix): REAL;
```

Fig. 19. Overloading operators for custom array types

3.3 Notes on Implementation

The implementation of the compiler modifications necessary for providing all language constructs presented in this paper is, at the time of submission, still work in progress. Nevertheless, in this paragraph we comment on some implementation specific details. For the built-in array types we decided – notionally - for a consistent memory representation that does not depend on array allocation kind such as dynamic, semi-dynamic, open or static. Since a special array is of value type, it at least consist of an array descriptor that includes the information about the array shape. An empty array `A` is characterized by `LEN(A)=0`. A schematic view of the memory structure is displayed in Fig. 20. The increment fields in the array descriptor are necessary for the sub-domain operations.

adr offset	description
...	type descriptor
+0	base address of data, points to *dataaddr* if array is on stack
+4	increment of dimension d-1
+8	length of dimension d-1
...	
$+8 \cdot (d-1) + 4$	increment of dimension 0
$+8 \cdot (d-1) + 8$	length of dimension 0
...	(padding)
+*dataaddr*	data base-address if array is on stack

Fig. 20. Schematic memory layout of built-in arrays

We give a short example of how range-valued indices are implemented: consider the assignment `A[a..b] := B[c,a..b]`. Both range valued indices `A[a..b]` and `B[c,a..b]` are of (one dimensional) array type; After range checks, corresponding increments $I_1 = \mathrm{Inc}(\mathtt{A}, 0)$ and $I_2 = \mathrm{Inc}(\mathtt{A}, 1)$, lengths $L_1 = L_2 = b - a + 1$, and base addresses $M_1 = \mathrm{Adr}(\mathtt{A}) + a \cdot \mathrm{Inc}(\mathtt{A}, 0)$ and $M_2 = \mathrm{Adr}(\mathtt{B}) + c \cdot \mathrm{Inc}(\mathtt{B}, 0) + a \cdot \mathrm{Inc}(\mathtt{B}, 1)$ are computed (according to the designators) and the two array descriptors are pushed on the stack. These are then used as arguments for the copy operation.

Note that it is not assumed that $\mathrm{Inc}(i) < \mathrm{Inc}(j)$ for $i < j$. This permits an arbitrary assignment of the contiguous part in the memory to a particular index, which is potentially useful for avoidance of cache missing. By introducing additional fields for an offset in each dimension, it would have easily, and without significant loss of efficiency, been possible to offer customizable lower bounds in arrays. However, for the given reasons (cf. Sect. 2), we decided against them.

Note that the displayed memory structure is only a very slight modification of the memory structure of classical arrays in Oberon. This can be regarded as confirmation of our maxims simplicity and efficiency.

4 Conclusion

The presented extension of the programming language Oberon is a further step in the direction of more intuitive and efficient mathematical programming. The introduction of a more flexible built-in array concept, including range-valued indices, leads to more compact and readable notation for computing with vectors, matrices etc. Being still safe it also has a high potential w.r.t. efficiency for the reasons of block-wise operations and the possible avoidance of cache missing. Flexibility and extensibility is granted by the introduction of custom array types that can be specified by the programmer and are syntactically compatible with the built-in array constructs.

References

1. R. Clint Whaley, Antoine Petitet. *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, Vol 35, No 2, (2005), pp. 101-121, John Wiley & Sons (2005)
2. R. Clint Whaley, Antoine Petitet, Jack J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Computing 27 (1-2), 2001, pp. 3-35, Elsevier Science Publishers B.V. (North Holland): Amsterdam-London-New York-Oxford-Paris-Shannon-Tokyo
3. Bradford L. Chamberlain, Lawrence Snyder, *Array language support for parallel sparse computation*, Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy, pp. 133 - 145 (2001), ACM Press New York
4. Specification of the Programming language *Chapel*, from `http://chapel.cs.washington.edu/`.
5. Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*, PhD thesis, University of Washington (2001).
6. Malik Silva and Richard Wait, *Cache Aware Data Layouts*, IITC2000, Colombo, January 2001
7. Malik Silva and Richard Wait, *Go for Both types of Data Locality!*, HPCAsia, Bangalore, December 2002
8. Alik Silva, *Sparse matrix storage revisited*, Proceedings of the 2nd conference on Computing frontiers, pp.230-235, Ischia, Italy (2005)
9. MATLAB documentation web site. `http://www.mathworks.com/access/helpdesk/help/techdoc/`
10. R Language Definition and Introduction on `http://www.r-project.org`.
11. Peter Januschke, *Oberon-XSC – Eine Programmiersprache und Arithmetikbibliothek für das Wissenschaftliche Rechnen*, PhD Thesis, Universität Karlsruhe, 1998.
12. Robert Griesemer, *A Programming Language for Vector Computers*, PhD Thesis, ETH Zürich, 1993.
13. Bernd Mösli,*A Comparison of C++, FORTRAN 90 and Oberon-2 for Scientific Programming*, GISI 95, editors: Friedbert Huber-Wäschle, Helmut Schauer, Peter Widmayer Berlin, Springer, p. 740-748, 1995.
14. Michael Baumgartner, *Erweiterung des Active Oberon Compilers und Software Entwicklungssystems im Hinblick auf die Ausnutzung der Intel SSE2 Vektoroperationen für mathematische Anwendungen*, Diploma Thesis, ETH Zürich, 2003.
15. Roberto Morelli, *Integration of OberonX in Oberon*, Diploma Thesis, ETH Zürich, 1997.
16. Günther Sawitzki, *Extensible Statistical Software: On a Voyage to Oberon*, J.Computational and Graphical Statistics, 5(3):263-283,1996.